Translation Certification for Smart Contracts

Jacco O.G. Krijnen[a,*], Manuel M. T. Chakravarty[b], Gabriele Keller[a], Wouter Swierstra[a]

[a] *Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, the Netherlands*
[b] *IOG Singapore Pte Ltd, 4 Battery Road, #25-01 Bank of China Building, Singapore*

Abstract

Compiler correctness is an old problem, but with the emergence of *smart contracts* on blockchains that problem presents itself in a new light. Smart contracts are self-contained pieces of software that control (valuable) assets in an adversarial environment; once committed to the blockchain, these smart contracts cannot be modified. Smart contracts are typically developed in a high-level contract language and compiled to low-level virtual machine code before being committed to the blockchain. For a smart contract user to trust a given piece of low-level code on the blockchain, they must convince themselves that (a) they are in possession of the matching source code and (b) that the compiler has correctly translated the source code to the given low-level code.

Classic approaches to compiler correctness tackle the second point. We argue that *translation certification* also squarely addresses the first. We describe the proof architecture of a translation certification framework and demonstrate how we can model the compilation pipeline as a sequence of translation relations. We give a detailed account of such relations for most passes of the Plutus Tx compiler, which we formalised in Coq. This approach facilitates a modular verification methodology and is robust in the face of an evolving compiler implementation.

Keywords: Compiler correctness, translation validation, Certified compilation, smart contracts

1. Introduction

Compiler correctness is an old problem that has received renewed interest in the context of *smart contracts*—that is, compiled code on public blockchains, such as Ethereum or Cardano. This code often controls a significant amount of financial assets, must operate under adversarial conditions, and can no longer be

[*] Corresponding author
Email addresses: j.o.g.krijnen@uu.nl (Jacco O.G. Krijnen), manuel.chakravarty@iohk.io (Manuel M. T. Chakravarty), g.k.keller@uu.nl (Gabriele Keller), w.s.swierstra@uu.nl (Wouter Swierstra)

updated once it has been committed to the blockchain. Bugs in smart contracts are a significant problem in practice [5]. Recent work has also established that smart contract language compilers can exacerbate this problem [28, Section 3], in this case, the Vyper compiler. More specifically, the authors report (a) that they did find bugs in the Vyper compiler that compromised smart contract security and (b) that they performed verification on generated low-level code, because they were wary of compiler bugs.

Hence, to support reasoning about smart contract source code, we need to get a handle on the correctness of smart contract compilers. On top of that, we do also need a *verifiable link* between the source code and its compiled code to prevent *code substitution attacks,* where an adversary presents the user with source code that doesn't match the low-level code committed on-chain.

In our previous work [20], we have reported on our ongoing effort to develop a certification engine for the open-source on-chain code compiler of the Plutus smart contract system[1] for the Cardano blockchain.[2] In this paper, we formally describe the *specification* of a significant part of the Plutus compiler, enabling us to reason formally about its behaviour. In particular, this paper describes two crucial aspects of our certification effort:

- We describe the architecture for a translation certifier based on *translation relations,* which allows us to generate *translation certificates*—proof objects that relate the source code to the resulting compiled code and to establish the correctness of the translation (Section 2).

- We provide formal definitions for the transformation passes that translate Plutus Intermediate Representation (PIR) to step-by-step Plutus Core (Section 3).

This paper elaborates on previous work [20] and makes the following novel contributions:

- Instead of discussing a simplified subset of PIR, we now cover the complete intermediate language and have modified the specification of the passes accordingly in Section 3. This required us to deal with more involved binding structures and types.

- We describe specifications of a few new compiler passes that were only recently implemented in the compiler or were not included in the previous work in Sections 3.3, 3.6, 3.7 and 3.8

- We provide an implementation of the translation relations in the Coq proof assistant.

[1] `https://developers.cardano.org/docs/smart-contracts/plutus/`

[2] `http://cardano.org` is, at the time of writing, the 5th largest public blockchain by market capitalisation.

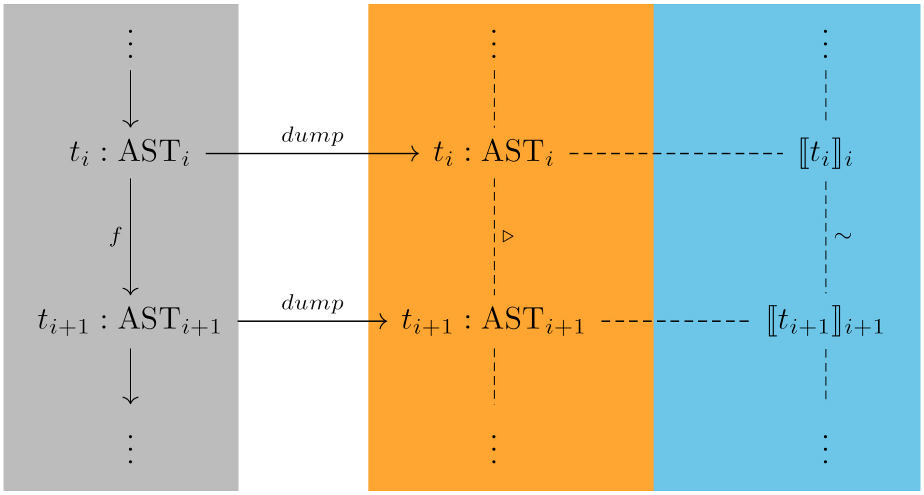

Figure 1: Architecture for a single compiler pass. The grey area (left) represents the compiler, orange (center) and blue (right) represent the certification component in Coq.

- We reflect on some practical proof engineering considerations that we encountered while implementing translation relations in the Coq proof assistant in Section 4.4.

2. The Architecture of the Certifier

On-chain code in the Plutus smart contract system is written in a subset of Haskell called *Plutus Tx* [18]. The Plutus Tx compiler is implemented as a plugin for the widely-used, industrial-strength GHC Haskell compiler, combining large parts of the GHC compilation pipeline with custom translation steps to generate Plutus Core. Unfortunately, it is infeasible to apply full-scale compiler verification à la CompCert [22], which was build from scratch with verification in mind, on existing, complex software such as GHC. We will therefore outline the design of a certification engine that, using the Coq proof assistant [6, 9], generates a proof object, or a *translation certificate*, asserting the validity of a Plutus Core program with respect to a given Plutus Tx source contract. In addition to asserting the correct translation *of this one program*, the translation certificate serves as a verifiable link between source and generated code.

We can view the compiler as a composition of pure functions that transform one abstract syntax tree (AST) into another. Figure 1 illustrates our certifier architecture for a single compiler pass, where the grey area represents the compiler implementation as series of functions $f : \mathrm{AST}_i \to \mathrm{AST}_{i+1}$. We use a family of types AST_i to illustrate that the representation of the abstract syntax might change after each transformation.

To support certification, the compiler outputs each intermediate AST so that we can parse these in our Coq implementation of the certifier. Within Coq, we define a high-level specification of each pass. We call such a specification a *translation relation*: a binary relation on abstract syntax trees that specifies the possible behaviour of the compiler pass. The orange area in Figure 1 displays the translation relation $\triangleright$ of one pass, where the vertical dashed line indicates that $t_i \triangleright t_{i+1}$ holds. To establish this, we define a decision procedure that, given two subsequent trees produced by the compiler, can find a proof.

The translation relation is purely syntactic—it does not assert anything about the correctness of the compiler—but rather *specifies* the behaviour of a particular compiler pass. To verify that the compilation preserves language semantics requires an additional proof, the blue area in Figure 1, that establishes that any two terms related by $\triangleright$ have the same semantics.

To illustrate our approach in this section, we will use an untyped lambda calculus, extended with non-recursive let-bindings.

$$t ::= x \mid \lambda x.\ t \mid t\ t \mid \texttt{let}\ x = t\ \texttt{in}\ t$$

In Section 3, we will consider full PIR (Plutus Intermediate Representation), which is a typed lambda calculus with many extensions.

2.1. Characterising a transformation

To assert the correctness of a single compiler pass $f : \mathrm{AST}_i \to \mathrm{AST}_{i+1}$, we begin by defining a translation relation $\triangleright$ on a pair of terms t_i and t_{i+1} which we call the "pre-term" and "post-term", respectively. This relation should syntactically characterise the admissible translations of that compiler stage, but it may be more general. In other words, $\triangleright$ should include the graph of f.

As a concrete example, consider an inlining pass. We have characterised this as an inductively defined relation in Figure 2. Here, $\Gamma \vdash s\ \triangleright\ t$ asserts that program s can be translated into t given an environment Γ of let-bound variables, paired with their definition. According to Rule [Inline-Var$_1$] the variable x may be replaced by t' when the pair (x, t) can be looked up in Γ and t can be translated to t', accounting for repeated inlining. The remaining rules are congruence rules, where Rule [Inline-Let] also extends the environment Γ. We omitted details about handling variable capture to keep the presentation simple: hence, we assume that variable names are globally unique.

Crucially, these rules do *not* prescribe which variable occurrences should be inlined, since the [Inline-Var$_1$] and [Inline-Var$_2$] rules overlap. The implementation of the pass may rely on a complex set of heuristics internal to the compiler. Instead, we merely define a relation capturing the *possible* ways in which the compiler *may* behave. This allows for a certification engine that is robust with respect to changes in the compiler, such as the particular heuristics used to decide whether to replace a variable with its definition or not.

We can then encode the relation $\cdot \vdash \cdot \triangleright \cdot$ in Coq as an inductive type `Inline`, which is indexed by an environment and two ASTs, as shown in Figure 3. This inductive type is a straightforward encoding of the rules of Figure 2: we define exactly one constructor per rule, and Γ is implemented as a cons-list.

These inductive types implement the translation relation: its inhabitants are proof derivations which will be a key ingredient of a compilation certificate.

$$\frac{\Gamma(x) = t \qquad \Gamma \vdash t \rhd t'}{\Gamma \vdash x \rhd t'}\ [\text{Inline-Var}_1]$$

$$\frac{}{\Gamma \vdash x \rhd x}\ [\text{Inline-Var}_2]$$

$$\frac{\Gamma \vdash t_1 \rhd t_1' \qquad (x, t_1), \Gamma \vdash t_2 \rhd t_2'}{\Gamma \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \rhd \textbf{let}\ x = t_1'\ \textbf{in}\ t_2'}\ [\text{Inline-Let}]$$

$$\frac{\Gamma \vdash t_1 \rhd t_1' \qquad \Gamma \vdash t_2 \rhd t_2'}{\Gamma \vdash t_1\ t_2 \rhd t_1'\ t_2'}\ [\text{Inline-App}]$$

$$\frac{\Gamma \vdash t_1 \rhd t_1'}{\Gamma \vdash \lambda x.t_1 \rhd \lambda x.t_1'}\ [\text{Inline-Lam}]$$

Figure 2: Characterisation of an inliner

```
Inductive Inline (Γ : list (string * term)) : term -> term -> Type :=
  | Inline_Var_1 : forall {x t t'},
      In (x, t) Γ ->
      Inline Γ t t' ->
      Inline Γ (Var x) t

  | Inline_Var_2 : forall {x},
      Inline Γ (Var x) (Var x)

  | Inline_Let : forall {x s t s' t'},
      Inline Γ s s' ->
      Inline ((x, s) :: Γ) t t' ->
      Inline Γ (Let x s t) (Let x s' t')

  | Inline_Lam : forall {x t t'},
      Inline Γ t t' ->
      Inline Γ (Lam x t) (Lam x t')

  | Inline_App : forall {s t s' t'},
      Inline Γ s s' ->
      Inline Γ t t' ->
      Inline Γ (App s t) (App s' t')
  .
```

Figure 3: Characterisation of an inliner in Coq

2.2. Decidability of translation relations

After defining a translation relation $\triangleright$ for a single compiler pass, we need a way to construct a proof that $t_i \triangleright t_{i+1}$ holds, for two particular terms t_i and t_{i+1}, produced by a run of the compiler.

We typically start by writing some derivation trees by hand for simple compilations using Coq's *tactics*. For straightforward relations, like the inline example sketched above, a proof can often be found with a handful of tactics such as `auto` or `constructor`. This is particularly useful as a simple way of testing the design of our relations. The drawback of this approach is, however, that it is difficult to reason when such a proof search may succeed or fail, or even terminate. Furthermore, the proof search quickly becomes slow for bigger ASTs and may result in large proof terms.

To address these issues, we write a semi-decision procedure in the style of ssreflect [17] of type

```
decide_inliner : term -> term -> bool
```

together with a proof

$$\text{sound} : \forall\ t\ t'.\ \texttt{decide\_inliner}\ t\ t' = \texttt{true} \rightarrow t \triangleright t'$$

which states that the decision procedure is sound with respect to the translation relation. Proofs can then be constructed with the proof term `sound t t' eq_refl`.

2.3. Verification

Given the relational specification $\triangleright$ of a compiler pass, we can now establish correctness properties of this pass. In the simplest case, this could be asserting the preservation of types. On the other end of the spectrum, we can demonstrate that related terms are semantically equivalent.

In Figure 1, we denote such correctness properties of $\triangleright$ in the blue area by means of an abstract binary relation $\sim$ on semantic objects $[\![\cdot]\!]$ of ASTs t_i. In the case of static semantics, we choose typing derivations as semantic objects, and (for most passes) syntactic equivalence of the types. The theorem then has the following form:

$$\text{preserves} : t_i \triangleright t_{i+1} \rightarrow (\Gamma \vdash t_i : \tau) \rightarrow (\Gamma \vdash t_{i+1} : \tau') \wedge \tau = \tau'$$

In the case of semantic equivalence, we define a logical relation for contextual equivalence [2], based on a big-step operational semantics of PIR. In this case, the semantic objects are (well-typed) terms, related by contextual equivalence: running any one-hole context with either of these terms embedded in it will result in equivalent termination behaviour. Note that such a proof requires well-typed terms, therefore relying on the aforementioned property of preservation of types.

Writing these proofs can be done independently and gradually (e.g. preservation of types first and contextual equivalence second) for each translation relation of the compiler pipeline. In fact, even without any formal verification of the translation relation, we can still provide some degree of confidence about the correctness of a compilation: one can inspect the (relatively concise) rules of a translation relation and run a decision procedure to confirm the terms of the compilation are related by it. After all, the translation relation is an independent specification of the admissible behaviour of the compiler pass.

This verification effort of translation relations is ongoing work and goes beyond the scope of this paper.

2.4. Certificate generation

This leads us to sketching the design of the certifier that generates certificates (or verifiable links) between the Plutus Intermediate Representation (PIR) and and the generated Plutus Core (PLC) target code. A certificate is a Coq proof script that is generated during the compilation of the code. First, it includes all intermediate ASTs $t_1, \ldots, t_n$, which the compiler emitted. Second, we relate every two subsequent ASTs in the appropriate translation relation $t_i \triangleright t_{i+1}$, by using the corresponding decision procedure and its soundness proof. Finally, we can include verification results: ideally this constitutes contextual equivalence proofs for each pass, which by transitivity show the semantic equivalence of source and target program.

Such a certificate can then be distributed alongside source code, giving the means for anyone to check it without having to trust the provider of the source code: they can inspect the involved ASTs, the translation relations and the theorems. Above all, the script can be run in the Coq kernel [9], to check the validity of the proofs. One can then be confident that the compiled code of the program is a faithful translation of the source code.

3. Translation Relations of the Plutus Tx Compiler

The Plutus Tx compiler translates Plutus Tx (a subset of Haskell) to Plutus Core, a variant of System F^{μ}_{ω} [13]. A hash of the Plutus Core code is committed to the Cardano blockchain, constituting the definitive reference to any deployed smart contract.

The Plutus Tx compiler reuses parts of the GHC infrastructure and implements its custom passes by installing a core-to-core pass plugin [15] in the GHC compiler pipeline. On a high level, the compiler comprises three steps:

1. The parsing, type-checking and desugaring phases of GHC are reused to translate a surface-level Haskell program into a GHC Core program.
2. A large subset of GHC Core is directly translated into an intermediate language named Plutus Intermediate Representation (PIR). These languages are similar and both based on System F, with some extensions. Additionally, the definitions of all referenced functions and types are included as local definitions so that the program is self-contained.
3. The PIR program is then transformed and compiled down into Plutus Core.

terms	t, u	$::=$	x	variable
			$\lambda x : T.t$	lambda abstraction
			$t\ t$	function application
			$\Lambda X :: K.t$	type abstraction
			$t\ \{T\}$	type application
			$\texttt{wrap}\ T\ U\ t$	wrap
			$\texttt{unwrap}\ t$	unwrap
			$\texttt{builtin}\ f$	built-in functions
			$\texttt{constant}\ k$	constant values
			$\texttt{error}\ T$	error
			$\texttt{let}\ [\texttt{rec}]\ \overline{b}\ \texttt{in}\ t$	let
bindings	b	$::=$	$x : T = t$	strict term binding
			$\sim x : T = t$	non-strict term binding
			$X :: K = T$	type binding
			$\texttt{data}\ X\ \overline{(Y :: K)} = \overline{c}\ \texttt{with}\ x$	datatype binding
constructors	c	$::=$	$x\ \overline{(T)}$	
types	T, U	$::=$	X	type variable
			$T \to U$	arrow type
			$\forall X :: K.T$	universal type
			$\lambda X :: K.T$	function type
			$T\ U$	function application
			$\texttt{builtin}\ C$	built-in types
			$\texttt{ifix}\ T\ U$	fixpoint type
kind	K	$::=$	$*$	type kind
			$K \Rightarrow K$	arrow kind
built-in functions	f	$::=$	$\ldots$	
constants	k	$::=$	$\ldots$	
built-in types	C	$::=$	$\ldots$	

Figure 4: Syntax of PIR and PLC, PIR-specific constructs are highlighted

The certification effort reported here focuses on Step 3, which is the most crucial component: it consists of multiple optimisations and translation schemes. PIR is a superset of the Plutus Core language: it adds several language constructs for the sake of convenience, such as user-defined datatypes, strict and non-strict let-bindings that may be (mutually) recursive. Some of the compilation steps translate these constructs into simpler language constructs.

In Figure 4 we present the syntax of PIR and Plutus Core, adapted from previous work [19]. The highlighted productions are specific to PIR, whereas the others are common to both languages. Expressions that have an overline, such as $\overline{(Y :: K)}$, should be read as any number of copies of that expression. [rec] indicates an optional occurrence of that keyword.

The first five term productions are familiar constructs of System F. The constructs wrap and unwrap form the isomorphism for iso-recursive types [19]. The let construct contains a group of bindings, which can be mutually recursive when the rec keyword is used. Otherwise, the bound names are scoped linearly. PIR supports several forms of bindings: strict and non-strict terms (indicated by a ∼ symbol before the variable name), types and algebraic datatypes with constructors and an eliminator.

The language of types again follows System F, but is extended with type abstraction and application for supporting higher kinds, as well as ifix for recursive types. Kinds are simple and can be either of function or base sort.

Finally, PIR comes with a set of built-in functions, constants and types. None of these play an important role in the translation relations, so we omit them for brevity. They include for example string and integer types with corresponding operations, as well as some cryptographic functions. In our Coq implementation, we have defined this grammar as a family of mutually recursive datatypes. We chose to represent variables as names, instead of the often used de Bruijn representation. We motivate this choice further in Section 4.4.4.

3.1. Example transformations

In Figure ?? we present a Haskell program to introduce some of the compiler passes that the Plutus Tx compiler performs. This program is a basic implementation of a *timelock*, a contract that states that funds may be moved after a certain date, or not at all. It contains a few contrived bindings (false and n') that will be useful to illustrate some transformations.

In figure **??**, we can see that the only occurrence of false has been inlined. Next, the dead code elimination pass cleans up the now unused definition of false. Finally, we can see how the definition of n' is floated up one level.

The above transformations are presented using the Haskell surface syntax, but in reality they happen on the PIR representation. Specifically, the code of Figure ?? is compiled to the following PIR program:

```
let data Bool = True | False with Bool_match in
let data Unit = Unit with Unit_match in
let nonrec strict lessThanEqInteger = ... in
data EndDate = Fixed Integer | Never with EndDate_match in
λ(end : EndDate) (current : Integer).
   let nonrec nonstrict false = False in
   EndDate_match end
```

```
-- | Either a specific end date, or "never".
data EndDate = Fixed Integer | Never

pastEnd :: EndDate -> Integer -> Bool
pastEnd end current =
  let false = False
  in case end of
    Fixed n ->  (let n' = if current >= 0 then n else 0 in n')
                   <= current
    Never   -> false
```

(a) Implementation of a time-lock

```
data EndDate = Fixed Integer | Never

pastEnd :: EndDate -> Integer -> Bool
pastEnd end current =
  let false = False
  in case end of
    Fixed n ->  (let n' = if current >= 0 then n else 0 in n')
                   <= current
    Never   -> False
```

(b) Result after inlining

```
data EndDate = Fixed Integer | Never

pastEnd :: EndDate -> Integer -> Bool
pastEnd end current =
  let false = False
  in case end of
    Fixed n ->  (let n' = if current >= 0 then n else 0 in n')
                   <= current
    Never   -> False
```

(c) Dead code, highlighted in gray

```
data EndDate = Fixed Integer | Never

pastEnd :: EndDate -> Integer -> Bool
pastEnd end current =
  in case end of
    Fixed n ->  let n' = if current >= 0 then n else 0 in
                n' <= current
    Never   -> False
```

(d) Result of let-floating

Figure 5: Code of a timelock, with several example compiler transformations

```
(λunit n . lessThanEqInteger
  (let nonrec nonstrict n' =
     Bool_match (greaterThanEqInteger current 0)
       (λunit . n) (λunit . 0)
       Unit
   in n')
  current)
(λunit . false)
Unit
```

Note that case distinction of a type T is encoded as the application of an eliminator function T_match, which is introduced as part of a data definition. Furthermore, the branches of a case distinction are delayed by abstracting over a unit value, since function arguments are evaluated strictly in PIR.

In the Appendix of previous work [20] we have described the result of each compiler passes on the above PIR program in detail.

3.2. Notational conventions

We will overload the $\triangleright$ symbol per sub-section that describes a translation relation. When necessary, we will disambiguate a specific relation with a subscript such as $\triangleright_{\textsc{Inline}}$. Furthermore, we may re-use the $\triangleright$ symbol for any of the different type of constructs in the AST grammar (terms, types, bindings).

Often, a translation relation is defined in some context that contains information about binders. We write Γ for contexts of term variables and Δ for contexts of type variables, and write the translation relation as $\Delta; \Gamma \vdash t \triangleright t'$. We model both of these contexts as ordered lists of pairs. We write $\Gamma(x) = y$ when (x, y) is the first occurrence of a pair in the list that has x as its first projection.

We sometimes write individual bindings from a `let` with square brackets like so: $[\sim x : \tau = t]$. When convenient, we omit type and kind annotations on binding sites like λ and `let`. Finally, We write `let` [`rec`] *bs* to describe a let binding that may be either recursive or non-recursive.

3.3. Properties of terms

The specification of some passes reuse a few common properties of programs. For example, the let-floating pass in Section **??** requires that the pre-term has globally unique variable names. We formalise such properties as inductive relations on a single abstract syntax tree.

3.3.1. Globally unique variables

For some passes it is assumed that variable names are globally unique, also known as the *Barendregt-convention.* It simply states that each variable name can only be bound once. We first define BoundIn, which relates a variable to a term in which it is bound.

$$\frac{}{\textsc{BoundIn}(x, \lambda x.\ t)}\ [\text{BoundIn-Lam-1}]$$

$$\frac{x \neq y \qquad \textsc{BoundIn}(x, t)}{\textsc{BoundIn}(x, \lambda y.\ t)}\ [\text{BoundIn-Lam-2}]$$

Other binding constructs such as `let` have analogous rules. The constructs in an AST that do not bind variable names simply require the property holds for each any sub-tree. We will sometimes also use FreeIn, which is defined similarly, but relates a variable to a term in which it occurs freely.

Then we define the global uniqueness property as a relation Unique:

$$\frac{\textsc{Unique}(t) \qquad \textsc{Unique}(\tau) \qquad \neg\textsc{BoundIn}(x, t)}{\textsc{Unique}(\lambda x : \tau.\ t)}\ [\text{Unique-Lam}]$$

Note that in the second hypothesis of [Unique-Lam], we have overloaded Unique for types, which is analogously defined with respect to type variable binders such as $\forall$.

3.3.2. Well-scoped expressions

Some of the compiler passes will reorder binders. In such cases, we require that the post-term is well-scoped. In other words, that the term contains no free variables. We define this property with an inductive well-scopedness relation $\Delta; \Gamma \vdash t$, which, similar to a typing relation, maintains a context Γ for term variables in scope, and Δ for type variables in scope.

$$\frac{\Delta; (x, \Gamma) \vdash t \qquad \Delta \vdash \tau}{\Delta; \Gamma \vdash \lambda(x : \tau).\ t}\ [\text{WellScoped-Lam}]$$

$$\frac{x \in \Gamma}{\Delta; \Gamma \vdash x}\ [\text{WellScoped-Var}]$$

Once again, this property extends to types with the relation $\Delta \vdash \tau$. We define the following abbreviation:

$$\textsc{Closed}(t) := \varepsilon; \varepsilon \vdash t$$

A closed term is well-scoped in the empty contexts.

3.3.3. Pure bindings

Several passes manipulate let bindings, but only when it safe to do so: special care has to be taken with strict bindings, which may diverge. Moving them may therefore change the meaning of a program.

For example, the following transformation is not semantics preserving, since the former term terminates, but the latter does not:

$$\lambda x.\ \texttt{let}\, y = \bot\, \texttt{in}\, 3 \;\rightarrow\; \texttt{let}\, y = \bot\, \texttt{in}\, \lambda x.\ 3$$

To ensure that let bindings are only transformed when it is safe to do so, the Plutus compiler tries to analyse if a strict binding is "pure", that is, the bound term terminates. Since this is of course undecidable, a few simple cases are considered, which we model in the relations Pure, PureBinding and PureBindings.

$$\frac{t \text{ is a value}}{\Gamma \vdash \textsc{Pure}(t)}$$

$$\frac{\Gamma(x) = \texttt{strict}}{\Gamma \vdash \textsc{Pure}(x)}$$

The relation $\Gamma \vdash \textsc{Pure}(t)$ classifies a subset of terms that are pure: those that are values, and variables that were bound strictly (meaning they are bound to a value). In this case, the environment Γ contains merely the annotations `strict` or `nonstrict` for each variable that is free in t. Note that this is relation does not have any recursive cases, and it relies on the environment Γ.

We then define PureBinding as follows:

$$\frac{}{\Gamma \vdash \textsc{PureBinding}({\sim}x = t)}$$

$$\frac{\Gamma \vdash \textsc{Pure}(t)}{\Gamma \vdash \textsc{PureBinding}(x = t)}$$

$$\frac{}{\Gamma \vdash \textsc{PureBinding}(\texttt{data}\ X\ (\overline{Y :: K}) = \overline{c}\ \texttt{with}\ x)}$$

$$\frac{}{\Gamma \vdash \textsc{PureBinding}(T :: K = \tau)}$$

The PureBindings relation then simply extends PureBinding to a binding group, requiring that all bindings are a PureBinding.

Although these relations require an environment Γ for strictness information about free variables, we will generally omit it in the presentation of translation relations, and write $\textsc{Pure}(t)$ for simplicity. In reality however, the rules of the translation relations also take care to construct the required context Γ.

3.4. Variable Renaming

The first compiler pass we present is the renaming pass. In this pass, the compiler transforms a program into an α-equivalent program, such that all variable names are globally unique, i.e. the post-term t satisfies $\textsc{Unique}(t)$. The implementation of some subsequent compiler passes depend on this property. We express variable renaming as a translation relation $\Delta; \Gamma \vdash t \ \triangleright\ t'$, stating that under the renaming environments Δ (for type-variables) and Γ (for term variables), t is renamed to t'. Both environments record the free variables, paired with their corresponding name in the post-term. Similarly, type-variables can be renamed, which we denote with $\Delta \vdash \tau \triangleright \tau'$.

The case for lambda abstractions is defined as follows:

$$\frac{\Delta; (x, y), \Gamma \vdash t \triangleright t' \qquad \Delta \vdash \tau \triangleright \tau' \qquad \textsc{NoCapture}(y, \Gamma, t)}{\Delta; \Gamma \vdash \lambda(x : \tau).\ t \triangleright \lambda(y : \tau').\ t'} \text{ [Rename-Abs]}$$

The [Rename-Abs] rule states that a lambda-bound variable x may be renamed at its binding-site to y, when t can be renamed to t' and τ to τ'. Of course, x may equal y, witnessing that no renaming took place. Lastly, we want to make sure not to relate invalid renamings, for example:

$$(\lambda x.\ \lambda z.\ x) \not\triangleright (\lambda y.\ \lambda y.\ y)$$

A straightforward way of enforcing this would be to add a hypothesis in [Rename-Abs] that $\forall v.\ (v, y) \notin \Gamma$, disallowing any shadowing in the post-term, but to remain as general as possible, we instead require that a new binder name y does not capture any free variable v in the pre-term that is also renamed to y in the post-term. Therefore, we define NoCapture as an implication:

$$\textsc{NoCapture}(y, \Gamma, t) := \forall v.\ (v, y) \in \Gamma \Rightarrow \neg\textsc{FreeIn}(v, t)$$

Rules of $\triangleright$ for for other binding constructs such as `let` or Λ are very similar.

The variable case simply follows from the environment Γ:

$$\frac{\Gamma(x) = y}{\Delta; \Gamma \vdash x \triangleright y} \text{ [Rename-Var]}$$

Note that in contrast to the Plutus Tx compiler, this translation relation does not establish global uniqueness of binders in the post-term, i.e. $t \triangleright t' \not\Rightarrow \textsc{Unique}(t')$. We consider that a specific property of the compiler implementation, allowing this renaming relation to be as general as possible.

Whenever a subsequent translation relation does require the Unique property on the pre-term, we will establish it separately by running the appropriate decision procedure.

3.5. Inlining

The rules of the translation relation for inlining in PIR are very similar to those of the untyped lambda calculus in Section 2.1. In addition, the Plutus Tx inliner also considers let-bound types $\texttt{let}\,\alpha :: K = \tau$, which may be inlined in type expressions. We therefore maintain a separate environment Δ of type bindings.

However, the Plutus Tx compiler does more than just inlining let-bound definitions. It also removes let-bindings that have been exhaustively inlined (also known as dead-code elimination) and it renames variables in inlined terms to preserve the property of global uniqueness. That is, we can model the pass as a composition of translation relations

$$\triangleright := \triangleright_{\text{RENAME}} \circ \triangleright_{\text{DEADCODE}} \circ \triangleright_{\text{INLINE}}$$

where $(R \circ S)(x, y) := \exists z. R(x, z) \wedge S(z, y)$.

This introduces a problem for our certification approach: we cannot observe and dump these "intermediate" ASTs, since they do not exist in the compiler! There, the three transformations are fused into a single pass.

To construct a proof relating two terms, then amounts to also finding the *intermediate term*, as part of the decision procedure. To simplify the search of these intermediate ASTs, we adapt the compiler to also emit supporting information about the performed pass; in this case, the compiler emits a list of the eliminated variables. If the compiler emits incorrect information, we may fail to construct a certificate, but we will never produce an incorrect certificate.

3.6. Beta redexes

Another obvious candidate for inlining is a beta-redex $(\lambda x.t_1)\ t_2$, which can be seen as another way of writing $\texttt{let}\ x = t_2\ \texttt{in}\ \ t_1$. Instead of changing the inlining pass described in sub-section **??**, the compiler has a small pass that rewrites such beta-redexes into (non-recursive) `let` constructs, after which the inlining pass takes care of the actual inlining.

More generally, the pass considers expressions of the form:

$$(\lambda x_1 \ldots x_n.t)\ t_1 \ldots t_n$$

It is important to notice that this is different from simply nesting normal beta redexes, which would look like this:

$$(\lambda x_1.\ \ldots ((\lambda x_n.t)\ t_n) \ldots)\ t_1$$

So in order to handle the former, we define a relation Betas to inductively relate such a term to a list of bindings bs and $t_{\texttt{in}}$, as they would appear in $\texttt{let}\ bs\ \texttt{in}\ t_{\texttt{in}}$ form. The key rule of the translation relation is then defined as:

$$\frac{\textsc{Betas}(t, bs, t_{\texttt{in}}) \qquad t_{\texttt{in}} \triangleright t'_{\texttt{in}} \qquad bs \triangleright bs'}{t \triangleright \texttt{let}\ bs'\ \texttt{in}\ \ t'_{\texttt{in}}}\ [\text{Betas}]$$

The pass does not only consider beta-redexes, but also instantiated type abstractions of the form $(\Lambda(\alpha :: \kappa).\ t)\ \{\tau\}$ which can similarly be treated as a let binding of a type and for which there is a rule analogous to [Betas] in the translation relation.

3.7. Splitting recursive let groups

Since the inlining pass does not consider bindings from a `let rec`, it is worth it to analyse whether such bindings are truly recursive. If not, any non-recursive bindings can be split out into a regular `let`, making them available for inlining. The compiler implements this pass by a strongly connected component analysis on the dependency graph obtained from the bindings.

We define this translation relation as follows:

$$t \triangleright t' := t \triangleright_s t' \wedge \textsc{Unique}(t) \wedge \textsc{Closed}(t')$$

This definition states that the pre- and post-term must be syntactically related, and the pre-term must have unique global binders, which means we do not have to worry about shadowing when reordering binders. Finally, the well-scopedness of the post-term ensures (in combination with $\textsc{Unique}(t)$) that any potential reordering of bindings is correct.

In order to define $\triangleright_s$ (s for syntactic), we first use a helper relation OuterBinds that can decompose a term of the following shape:

$$\texttt{let}\ [\texttt{rec}]\ bs_1\ \texttt{in}\ \dots\ \texttt{let}\ [\texttt{rec}]\ bs_n\ \texttt{in}\ t_{in}$$

The relation $\textsc{OuterBinds}(t, [bs_1,\ \dots\ , bs_n], t_{in})$ holds if term t has that shape.

Then we define $\triangleright_s$ in the `let rec` case as:

$$\frac{\textsc{OuterBinds}(t_{post}, bs', t'_{in}) \qquad bs \triangleright_s bs' \qquad t_{in} \triangleright_s t'_{in}}{\texttt{let rec}\ bs\ \texttt{in}\ t_{in} \triangleright_s t_{post}}\ [\textsc{Split}\text{-let-rec}]$$

The rule [Split-let-rec] states that if bs' are outer bindings in t_{post}, and they are related with bs and the let body t_{in} is related with the inner let body in t_{post}, the terms are related. Any scoping related concerns are once again dealt with by requiring Unique on the pre-term and Closed on the post-term.

3.8. Unwrap-wrap elimination

After inlining, it can happen that some expressions can be simplified. The unwrap-wrap pass cleans up a specific artifact which may appear:

$$\frac{t\ \triangleright\ t'}{\texttt{unwrap}\ (\texttt{wrap}\ T\ U\ t) \triangleright t'}\ [\text{Unwrap-Wrap}]$$

The `wrap` and `unwrap` constructs form the isomorphism of iso-recursive types [19], hence their composition is the identity.

3.9. Dead code elimination

By means of a live variable analysis, the compiler determines which let-bound definitions are unused and can be removed. This is often useful for definitions that are introduced by other compiler passes. Since PIR is a strict language, however, the compiler can only eliminate those bindings for which it can determine they are a PureBinding, otherwise a diverging program may suddenly become terminating.

The analysis in the compiler is not as straightforward as counting occurences. Even a let-bound variable that *does* occur in the code, may be dead code, when it is only used in other dead bindings. This is also known as strongly live variable analysis [16].

In the translation relation we require that binders in the pre-term are unique, and that the post-term is well-scoped. We will then define $\triangleright_s$, that characterises the removal of bindings.

$$t \triangleright t' := \textsc{Unique}(t) \wedge \textsc{Closed}(t') \wedge t \triangleright_s t'$$

Let us first consider the case of $\triangleright_s$ where a complete `let` has been eliminated:

$$\frac{\textsc{PureBindings}(bs) \qquad t \triangleright_s t'}{\texttt{let}\ [\texttt{rec}]\ bs\,\texttt{in}\,t\ \triangleright_s\ t'}\ \text{[DeadBindings-Let-1]}$$

Given that all bindings are pure, they can be removed. Note that we do not mention any conditions about whether the bindings are actually dead code: this is covered by the requirement that the pre-term is Unique and post term is Closed. We elaborate on why both conditions are necessary in Section **??**

This pass may also eliminate some, but not all bindings in a `let`. We treat that as a different case:

$$\frac{\begin{array}{c}\forall b \in bs.\ \textsc{Removed}(b, bs') \Rightarrow \textsc{PureBinding}(b)\\ \forall b' \in bs'.\ \exists b \in bs.\textsc{Name}(b') = \textsc{Name}(b) \wedge b\ \triangleright_s\ b'\\ t \triangleright_s t'\end{array}}{\texttt{let}\ [\texttt{rec}]\ bs\,\texttt{in}\,t \triangleright_s \texttt{let}\ [\texttt{rec}]\ bs'\ \texttt{in}\ t'}\ \text{[DeadBindings-Let-2]}$$

The first hypothesis states that any binding (identified by its unique name) which is not present in the binding group of the post-term must be a PureBinding. Removed is defined by simply comparing binders by name, which are globally unique. Second, we require that any binding in the post-term has a related binding in the pre-term. These two conditions imply that the bindings of the post-term form a subset of those in the pre-term (allowing for potential reordering). Lastly, the let bodies must also be related.

3.10. Let-floating

During let-floating, let-bindings can be moved upwards in the program. This may save unnecessarily repeated computation and makes the generated code more readable. The Plutus Tx compiler constructs a dependency graph to maintain a correct ordering when multiple definitions are floated. For the translation relation, we first consider the interaction of a `let` expression with its parent node in the AST. For example, consider the case of a lambda with a `let` directly under it:

$$\frac{\textsc{PureBindings}(bs)}{\begin{array}{c}\lambda x.\ \texttt{let}\ [\texttt{rec}]\ bs\ \texttt{in}\ t\\ \triangleright_s\\ \texttt{let}\ [\texttt{rec}]\ bs\ \texttt{in}\ (\lambda x.\ t)\end{array}}\ \text{[Float-Let-Lam]}$$

This rule states that a (possibly recursive) binding group consisting of only pure bindings may float up past a lambda. This restriction is necessary for preserving termination behaviour. We use the operator $\triangleright_s$ to denote that this is the "syntactic" part of the translation relation, and define the full relation $\triangleright$ below. Similar rules express how a `let` can float past other language constructs. Since the compiler pass may float `let`s more than just one step up, we use the transitive closure $\triangleright_s^+$ as part of the final definition.

Furthermore, the pass may reorder bindings within a binding group ($\triangleright_{\textsc{Reorder}}$), and combine the bindings of adjacent `let` groups into a single group ($\triangleright_{\textsc{Merge}}$). We omit the details of these relations , as the former is defined similarly to the reordering that can occur in dead-code elimination (Section **??**), and the latter is based on OuterBinds, as described in Section **??**.

The complete translation relation of this pass is then defined as:

$$t \triangleright t' := (\triangleright_{\textsc{Merge}} \circ \triangleright_{\textsc{Reorder}} \circ \triangleright_s^+)(t,\ t') \wedge \textsc{Unique}(t) \wedge \textsc{Closed}(t')$$

Note that we do not need to maintain a dependency graph in the certifier, but only need to assert that transformations do not break dependencies.

3.11. Encoding of non-strict bindings

The PIR language has both strict and non-strict let-bindings, but Plutus Core does not. The *thunking transformation* is used to eliminate non-strict let-bindings, by encoding them as strict bindings. We define the rules as a relation $\Gamma \vdash t \triangleright t'$, where Γ records for every bound variable whether it was bound `strict` or `nonstrict`. We first consider how a non-recursive, non-strict binding is translated:

$$\frac{\Gamma \vdash t_1 \triangleright t_1' \qquad (x, \texttt{nonstrict}), \Gamma \vdash t_2 \triangleright t_2' \qquad \neg\textsc{FreeIn}(y, t_1)}{\Gamma \vdash [\sim x = t_1 \ \texttt{in} \ t_2] \triangleright [x = \lambda(y : \texttt{()}).\ t_1' \ \texttt{in} \ t_2']} \text{ [Thunk-NR-NS]}$$

This rule states that the bound term is thunked by introducing a lambda abstraction that expects a value y of unit type as its argument. The rule for a *recursive* let-binding is very similar, but also extends the environment under which t_1 is transformed.

Finally, we also have to replace the occurrences of these non-strict variables, adding an application to the unit value `()`, thereby forcing evaluation.

$$\frac{\Gamma(x) = \texttt{nonstrict}}{\Gamma \vdash x \triangleright x \ \texttt{()}} \text{ [Thunk-Var]}$$

3.12. Thinking of recursive bindings

This pass changes strict bindings in a `let rec` to non-strict bindings, which are then directly forced again by the addition of a strict binding with the same name, in order to preserve termination behaviour. For example:

$$\texttt{let}\,\texttt{rec}\ (x : \tau) = t_1\,\texttt{in}\,t_2 \ \triangleright\ \texttt{let}\,\texttt{rec}\ \sim(x : \tau) = t_1\,\texttt{in}\,\texttt{let}\ (x : \tau) = x\,\texttt{in}\,t_2$$

The point of this tranformation is that the thunking transformation (Subsection 3.11), which runs after this pass, will translate the non-strict binding of x back into a strict (but now thunked) form:

$$\texttt{let}\,\texttt{rec}\ (x : \texttt{()} \to \tau) = t_1\,\texttt{in}\,\texttt{let}\ (x : \tau) = x\ \texttt{()}\,\texttt{in}\,t_2$$

This combination of two passes establishes the property that all recursive bindings are of *function type*, which is a requirement for the compilation of `let rec` (Subsection **??**).

We capture the first part of this transformation in a rule for a strict binding:

$$\frac{t \triangleright t'}{[x = t] \triangleright [\sim x = t'] \mid \{x\}}\ \text{[Thunk-TermBind-1]}$$

When relating individual bindings, we write $b \triangleright b' \mid V$. Here the set V contains those variables that will need additional (shadowing) strict bindings in the post-term. In the above rule, this is a singleton set with x. This set may be empty when bindings do not change their associated strictness:

$$\frac{t \triangleright t'}{[x = t] \triangleright [x = t'] \mid \emptyset}\ \text{[Thunk-Bind-Cong]}$$

In fact, when strictness *is* changed, but the bound term is known to be Pure (i.e. terminating), the compiler does not introduce the additional strict counterpart, since termination behaviour will not change. For this case we have an additional rule which again has the empty set for V:

$$\frac{t \triangleright t' \qquad \textsc{Pure}(t)}{[x = t] \triangleright [\sim x = t'] \mid \emptyset}\ \text{[Thunk-TermBind-2]}$$

The case of $\triangleright$ for `let` now becomes:

$$\frac{bs \triangleright bs' \mid V \qquad t \triangleright t' \qquad bs_V = \{[v = v] \mid v \in V\}}{\texttt{let}\,\texttt{rec}\ bs\ \texttt{in}\,t \triangleright \texttt{let}\,\texttt{rec}\ bs'\ \texttt{in}\ (\texttt{let}\ bs_V\ \texttt{in}\ t')}\ \text{[Thunk-Let-Rec]}$$

Here, the first hypothesis relates the bindings in bs and bs' point-wise, where V is the union of all their sets of variables. The bindings bs_V in the post-term should then be exactly the strict bindings for the variables in V.

3.13. Further passes

There are two other passes in the PIR-to-PLC pipline: the compilation of (mutually) recursive let-bindings and compilation of algebraic datatypes.

- The compilation of recursive binding groups is achieved by encoding them as non-recursive lets, which happens in two conceptual steps: first the group is converted into a single (recursive) binding of tuple type, where each component corresponds to one of the original bindings. Second, a fixpoint combinator is introduced, specific to the size of the tuple. This fixpoint is then used to translate the recursively bound tuple into a non-recursive binding. The original names of the binding group are then simply projected out of the tuple.

- The compilation of algebraic datatypes considers `data` definitions, such as:

  $$\texttt{let data}\ \mathit{Maybe}\ \alpha = \mathit{Just}\ \alpha \mid \mathit{Nothing}\ \texttt{with}\ \mathit{maybe}\ \texttt{in}\ t$$

  By means of the Scott encoding [1], they are transformed to a term that uses type and term abstractions with equivalent definitions, of the form:

  $$(\Lambda \mathit{Maybe}.\lambda \mathit{Just}.\lambda \mathit{Nothing}.\lambda \mathit{maybe}.\ t')\ \tau_{\mathit{Maybe}}\ t_{\mathit{Just}}\ t_{\mathit{Nothing}}\ t_{\mathit{maybe}}$$

Both of these passes have already been described in great detail elsewhere [19]. They rely on the Scott encoding for handling algebraic datatypes (in the first case: Scott-encoded tuples), but that approach will likely change soon with the introduction of native sum and product types in PLC[3]. We have therefore not formalised these passes.

3.14. Encoding of non-recursive bindings

At this point in the compiler pipeline, there is only one type of `let` construct that may still occur: strict, non-recursive bindings. Such bindings are simply compiled into a repeated β redexes:

$$\frac{t_1 \triangleright t_1' \ \ \ldots\ \ t_n \triangleright t_n' \qquad t \triangleright t'}{\texttt{let}\, x_1 = t_1 \ \ \ldots\ \ x_n = t_n\, \texttt{in}\, t \ \triangleright\ (\lambda x_i \ \ \ldots\ \ x_n.\, t')\ t_1' \ \ \ldots\ \ t_n'}\ \text{[Redex-Let]}$$

4. Engineering considerations

In this section, we evaluate our approach to certifying an independently developed, constantly evolving compiler under the application constraints imposed by smart contracts. We also describe lessons learnt regarding the architecture of proofs, such that they are robust and maintainable.

4.1. Gradual verification

The certifier architecture outlined in this paper allows for a gradual approach to verification: during the development of the certification engine, each individual step in the process increases our overall confidence in the compiler's

[3] url to github proposal

correctness, even if we have not yet completed the end-to-end semantic verification of the compiler pipeline.

By defining only the translation relations, we have an independent formal specification of the compiler's behaviour. This makes it easier to reason informally and to spot potential mistakes or problems with the implementation.

Implementing the decision procedures for translation relations ties the implementation to the specification: we can show on a per-compilation basis that a pass is sound with respect to its specification as a translation relation. Furthermore, we can test that these translation relations accurately model the compiler's behaviour by automatically constructing evidence for various input programs, such as those contained in the testsuite of the compiler.

Finally, by proving semantics preservation of a translation relation, we establish that the corresponding pass of the compiler is correct; for each of run of the compiler for which we can establish that the translation relation holds, we know that the semantics of the pre-term and post-term coincide.

4.2. Agility

The Plutus Tx compiler is developed independently of our certification effort. Moreover, it relies on a substantial existing code base—namely, that of the Glasgow Haskell Compiler (GHC). In addition, both GHC and the Plutus Tx-specific parts evolve on a constant basis, improving code generation or fixing bugs.

Given these constraints, full verification of the compiler appears to be in conflict with the ongoing maintenance of the compiler. A proof on the basis of the compiler source code would constantly have to adapt to the evolving compiler source. Hence, the architecture of our certification engine is based on a *grey box approach*, where the certifier matches the general outline (such as the phases of the compiler pipeline), but not all of the implementation details of the compiler. For example, our translation relation for the inliner admits any valid inlining. Any changes to the compiler's heuristics to produce more efficient programs by being selective about what precisely to inline do not affect the inliner's translation relation, and hence, do not affect the certifier.

4.3. Trusted Computing Base (TCB)

The fact that the Plutus Tx compiler is not implemented in a proof assistant, but in Haskell complicates direct compiler verification. It might be possible to use a tool like hs-to-coq [31], which translates a subset of Haskell into Coq's Gallina and has been used for proving various properties about Haskell code [11]. However, given that those tools often only cover language subsets, it is not clear that they are applicable. More importantly, such an approach would increase the size of the trusted computing base (TCB), as the translation from Haskell into Coq's Gallina is not verified. Similarly, extraction-based approaches suffer from the same problem if the extraction itself is not verified, although there are projects like CertiCoq [3] that try to address that issue.

In any case, our architecture has a relatively small TCB. We directly relate the source and target programs via a chain of intermediate ASTs, taking the compiler implementation out of the equation. Trusting a translation certificate comes down to trusting the Coq kernel that checks the proof, the theorem with its supporting definitions and soundness of the Plutus Core interpreter with respect to the formalised semantics. Of course, these components are part of the TCB of a verified compiler too. This aspect also motivated our choice of Coq over other languages such as Agda, due to its relatively small and mature kernel.

4.4. Proof architecture

Since the original implementation [20], we have extended and revised several of the implementations of the translation relations. We describe some insights obtained in the process.

4.4.1. Composed translation relations

A convenient pattern that we have found is to define some relations as a conjunction of simpler relations and predicates, such as the translation relation for splitting recursive let groups (Section 3.7):

$$t \triangleright t' := t \triangleright_s t' \wedge \textsc{Unique}(t) \wedge \textsc{Closed}(t')$$

Here we split the relation into a syntactic transformation, $t \triangleright_s t'$, and various side conditions. The syntactic transformation is easy to specify; in contrast, a transformation that would also guarantee to preserve scoping of the post-term would be significantly harder to realise. Instead, we require the Unique and Closed conditions separately, ensuring that bindings in a `let` binding group may be reordered arbitrarily in a compiler pass as long as these separate conditions hold.

Another case where this split of the relation is useful, is the dead code elimination pass (Section **??**). The syntactic rules merely assert that only pure bindings can be removed, Unique and Closed predicates ensure that only dead code is removed. Note that for dead code, $\textsc{Closed}(t')$ on its own is not a sufficient condition. For example, consider the following program, where shadowing occurs:

$$\texttt{let}\, x = \texttt{true}\,\texttt{in}\,\texttt{let}\, x = \texttt{false}\,\texttt{in}\, x$$

Removing the second binding of x results in a Closed term, but is not a safe transformation! Admittedly, the translation relation could be slightly more general by requiring that each eliminated binding was not shadowing other bindings. However, we prefer our formulation as a conjunction of three simple properties, because it keeps the rules of $\triangleright_s$ straightforward due to avoiding the need for an additional context or non-local information about variable bindings. Furthermore, it still accurately describes the compiler's implementation, which also assumes global uniqueness on the pre-term.

4.4.2. Reducing boilerplate rules

Many inductive translation relations contain boilerplate constructors; typically there are only a handful of interesting rules, and all other AST constructs correspond to congruence rules, specifying the translation relation contains the identity relation.

In our Coq implementation, we factor out this boilerplate by defining a separate congruence relation, Cong. This is a ternary relation, between some translation relation $\triangleright$ and two terms, that requires that each construct in the AST that the direct sub-terms to be related by $\triangleright$. For example, in the case of function application, we have:

$$\frac{s \triangleright s' \qquad t \triangleright t'}{\textsc{Cong}(\triangleright, s\ t, s'\ t')}\ [\textsc{Cong}\text{-App}]$$

The use of Cong simplifies some relations such as dead-code elimination (Section **??**) or splitting recursive binding groups (Section **??**), but is not general enough for relations that use environments Γ and Δ with binder-specific information, such as renaming. In the future, we may extend these congruences with more arguments, enabling further reuse.

4.4.3. Relational versus functional specifications

Often, we prefer relational specifications over functions, since they are more general and can express many-to-many relations, such as in the specification of inlining (Section **??**). On the other hand, some passes do not need this level of generality and we can model that pass as a total Coq function. The benefit of doing so is that a functional specification does not require a separate decision procedure: by simply running the function on the pre-term and syntactically comparing its output with the post-term we can establish the validity of the translation.

Therefore, we have sometimes chosen to formalise the specification as a function. One example is the encoding of non-recursive bindings (Section **??**), which uses a clear translation scheme that applies to *every* non-recursive `let` definition. We have (partially) implemented that pass as a Coq function:

```
Fixpoint compile_term : Term -> Term
```

Similarly, the compilation of recursive binding groups can be implemented as a Coq function `encode_let_rec`. However, we noticed that it is cumbersome to try and replicate the compiler's strategy of generating fresh variables. Therefore, we combined this functional specification with the relational specification of renaming of Section **??**:

$$t \triangleright t' := (\texttt{encode\_let\_rec}\ t)\ \triangleright_{\textsc{Rename}}\ t'$$

4.4.4. Variable representation

A common way of representing variables is by using de Bruijn indices. However the Plutus compiler uses named variables. Specifically, a variable is a pair of type String × $\mathbb{N}$, where the first component is the name of the variable as it appears in the source code and the second is the identifier used internally by the compiler. The compiler goes through quite some effort to keep the natural number globally unique, as we discussed in Section **??**.

Our Coq AST type is flexible and parametrised by the type that is used for binders. This allows us some more freedom for experimentation, but at the moment, we have only instantiated this with (globally unique) String identifiers. We plan to change this to match the compiler implementation in the near future, which should be a straightforward refactoring.

By design, we are staying close to the compiler's internal representation of ASTs: we want to specify *syntactic* translation relations that stay as close as possible to the compiler's behaviour. Any more abstract notion of binding — such as well-typed or well-scoped de Bruijn indices — would require additional checks and conversions of each intermediate AST. Furthermore, we would have to trust that the static or dynamic semantics that we formalise correspond to the semantics of the original form.

4.4.5. "Big-step" versus transitive closure of "small-step"

Most translation relations are inductive: they have hypotheses that require sub-trees to be related inductively. In the case of let-floating (Section **??**), we used a different approach, since it is not a local transformation. Instead, we first specified a single "floating" step, and then took the transitive closure of that relation to describe the entire the pass. In fact, many pass specifications could be described in that way but we foresee that this approach can become problematic when defining the corresponding decision procedure: we effectively require a (potentially long) list of intermediate AST's, which are not all emitted by the compiler. For that reason, we prefer inductively defined relations.

4.4.6. Deriving sound decision procedures

Since we are currently writing decision procedures for translation relations by hand, we are investigating ways to automate this process. In many cases one can "read of" a (naive) strategy for a decision procedure from the rules of a relation. Indeed, recent work [27] by Paraskevopoulou *et al.* has shown how to derive computational content, including verified decision procedures, from Coq's inductive definitions. We hope that by building on this work, we can lower the (maintenance) cost associated with the specification of a compiler pass even further.

5. Related Work

5.1. Compilers and correctness

The standard approach to compiler correctness is *full compiler verification*: a proof that asserts that the compiler is correct as it demonstrates that, for any valid source program, the translation produces a semantically equivalent target program. Examples of this approach include the CompCert [22] and CakeML [21] projects, showing that (with significant effort) it is possible to verify a compiler end-to-end. To do so, the compiler is typically implemented in a language suitable for verification, such as the Coq proof assistant or the HOL theorem prover.

In contrast, the technique that we propose for the Plutus Tx compiler is based on *translation validation* [29]. Instead of asserting an entire compiler correct, translation validation establishes the correctness of individual compiler runs.

A statement of full compiler correctness is, of course, the stronger of the two statements. Translation validation may fail to assert the correctness of some compiler runs; either because the compiler did not produce correct code or because the translation certifier is incomplete. In exchange for being the weaker property, translation validation is potentially (1) less costly to realise, (2) easier to retrofit to an existing compiler, and (3) more robust in the face of changes to the compiler.

The Cogent certifying compiler [26] has shown that it is possible to use translation validation for lowering the cost of functional verification of low-level code: a program can be written and reasoned about in a high-level functional language, which is compiled down to C. The generated certificate then proves a refinement relation, capable of transporting the verification results to the corresponding C code. The situation is different from ours: the Cogent compiler goes through a range of languages with different semantic models and uses the forward-simulation technique as a consequence. In contrast, we are working with variations of lambda calculi that have similar semantics, allowing us to use logical relations and translation relations.

The idea of *proof-carrying code* [24] is closely related to translation validation, shifting the focus to compiled programs, rather than the compiler itself. A program is distributed together with a proof of a property such as memory or type safety. Such a proof excludes certain classes of bugs and gives direct evidence to the users of such a program, who may independently check the proof before running the program. Our certification effort, while related, differs in that we keep proof and program separate and in that we are interested in full semantic correctness and not just certain properties like memory and type safety.

In their Coq framework [23], Li and Appel use a technique similar to ours for specifying compiler passes as inductive relations in Coq. Their tool reduces the effort of implementing program transformations and corresponding correctness proofs. The tool is able to generate large parts of an implementation together with a partial soundess proof with respect to those relations. The approach is used to implement parts of the CertiCoq backend.

5.2. Certificates and smart contracts

Smart contracts often manage significant amounts of financial and other assets. Before a user engages with such a contract, which has been committed to the blockchain as compiled code, they may want to inspect the source code to assert that it behaves as they expect. In order to be able to rely on that inspection, they need to know without doubt that (1) they are looking at the correct source code and (2) that the source code has been compiled correctly.

While a verified smart contract compiler addresses the second point, it doesn't help with the first. An infrastructure of *reproducible builds*, on the other hand, solves only the first point. The latter is the approach taken by Etherscan[4]: to verify that a deployed Ethereum smart contract was the result of a compiler run, one provides the source code and build information such as the compiler version and optimisation settings.

In contrast, a *certifying compiler* [25] that generates an independently verifiable certificate of correct translation, squarely addresses both points. By verifying a smart contract's translation certificate, a smart contract user can convince themselves that they are in possession of the matching source code and that this was correctly compiled to the code committed to the blockchain.

5.3. Verification in the smart contract domain

Ethereum was the first blockchain to popularise use of smart contracts, written in the Solidity programming language. Solidity is an imperative programming language that is compiled to EVM bytecode, which runs on a stack machine operating on persistent mutable state. The DAO vulnerability [12] has underlined the importance of formal verification of smart contracts. Notably, a verification framework has been presented [10] for reasoning about embedded Solidity programs in F*. The work includes a decompiler to convert EVM bytecode, generated by a compiler, into Solidity programs in F*. The authors propose that correctness of compilation can be shown by proving equivalence of the embedded source and (decompiled) target program using relational reasoning [7]. However, this would involve a manual proof effort on a per-program basis, and relies on the F* semantics since the embeddings are shallow. Furthermore, components such as the decompiler are not formally verified, adding to the size of the TCB.

The translation validation technique has been used for the verification of a particular critical Ethereum smart contract [28] using the K framework. The work demonstrates how translation validation can succesfully be applied to construct proofs about the low-level EVM bytecode by mostly reasoning on the (much more understandable) source code. The actual refinement proof is still constructed manually, however.

The Tezos blockchain also uses a stack-like language, called Michelson. The Mi-Cho-Coq framework [8] formalises the language and supports reasoning with a weakest precondition logic. There is ongoing work for developing a certified compiler in Coq for the Albert intermediate language, intended as a target language for certified compilers of higher-level languages. This differs from our approach as it requires the compiler to be implemented in the proof assistant.

[4] https://etherscan.io/verifyContract

ConCert is a smart contract verification framework in Coq [4]. It enables formal reasoning about the source code of a smart contracts, defined in a different (functional) language. The programs are translated and shallowly embedded in Coq's Gallina. Interestingly, the translation is proven sound, in contrast with approaches such as hs-to-coq [31], since it is implemented using Coq's metaprogramming and reasoning facility MetaCoq [30].

6. Conclusions and further work

The Plutus Tx compiler translates a Haskell subset into Plutus Core. The compiler consists of three main parts: the first one reuses various stages of GHC to compile the Haskell subset to GHC Core—the principal intermediate language used by GHC. The second part translates GHC Core to PIR and the final part compiles PIR to Plutus Core. As Plutus Core is strict and does not directly support datatypes, the corresponding translation scheme is quite complex. Moreover, it contains numerous transformation and optimization passes.

In this paper, we focused on the certification effort covering the third part of that pipeline; specifically, the translation steps from PIR to Plutus Core. While the other passes are certainly important for end to end verification, they are largely syntactic transformations that desugar user defined programs to a smaller core language. We have developed translation relations for all passes described in Section 3, such that we can, for example, produce a proof relating the previously described timelock example in PIR to its final form in Plutus Core. For some of these passes, such as inlining, we have implemented a verified decision procedure, but most of the evidence is generated semi-automatically by using Coq tactics. We have not yet covered all transformations in their full generality; for example, we do not cover (mutually) recursive datatypes yet. We have also started the semantic verification of key passes of the translation[14] and are investigating different ways to automate decision procedures for larger programs effectively.

In the future, we hope to continue this line of work in four directions: (1) filling in the remaining gaps in translation relations (such as covering mutually recursive datatypes); (2) completing the decision procedures associated with each translation relation; (3) continuing the semantic verification of the compiler passes; and (4) further automating our approach and improve the efficiency and maintainability of the certifier.

7. Acknowledgements

Funding for this research was provided by NWO and IOG. Furthermore, we would like to thank Michael Peyton Jones for his feedback and Joris Dral for his contributions in the Coq implementation.

References

[1] Abadi, M., Cardelli, L., Plotkin, G.: Types for the Scott numerals (1993)

[2] Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: European Symposium on Programming. pp. 69–83. Springer (2006)

[3] Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: The third international workshop on Coq for programming languages (CoqPL) (2017)

[4] Annenkov, D., Nielsen, J.B., Spitters, B.: ConCert: a smart contract certification framework in Coq. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 215–228 (2020)

[5] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Principles of Security and Trust (POST 2017). LNCS, vol. 10204 (2017)

[6] Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al.: The Coq proof assistant reference manual: Version 6.1. Ph.D. thesis, Inria (1997)

[7] Barthe, G., Fournet, C., Grégoire, B., Strub, P.Y., Swamy, N., Zanella-Béguelin, S.: Probabilistic relational verification for cryptographic implementations. ACM SIGPLAN Notices **49**(1), 193–205 (2014)

[8] Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-Cho-Coq, a framework for certifying Tezos smart contracts. In: International Symposium on Formal Methods. pp. 368–379. Springer (2019)

[9] Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)

[10] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM workshop on programming languages and analysis for security. pp. 91–96 (2016)

[11] Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J., Weirich, S.: Ready, set, verify! applying hs-to-coq to real-world Haskell code (experience report). Proceedings of the ACM on Programming Languages **2**(ICFP), 1–16 (2018)

[12] Buterin, V.: CRITICAL UPDATE Re: DAO Vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/ (2016), retrieved December 10, 2021

[13] Chapman, J., Kireev, R., Nester, C., Wadler, P.: System F in Agda, for fun and profit. In: Mathematics of Program Construction (MPC 2019). LNCS, vol. 11825 (2019)

[14] Dral, J.: Verified Compiler Optimisations. Master's thesis, Utrecht University (2022)

[15] GHC Team: GHC 9.0 User Manual. https://downloads.haskell.org/~ghc/9.0.1/docs/html/users_guide/extending_ghc.html

[16] Giegerich, R., Möncke, U.: Invariance of approximative semantics with respect to program transformations. In: GI—11. Jahrestagung, pp. 1–10. Springer (1981)

[17] Gonthier, G., Le, R.S.: An Ssreflect Tutorial. Ph.D. thesis, INRIA (2009)

[18] IOHK: The Plutus Platform and Marlowe 1.0.0 documentation. https://plutus.readthedocs.io/en/latest/plutus/tutorials/plutus-tx.html

[19] Jones, M.P., Gkoumas, V., Kireev, R., MacKenzie, K., Nester, C., Wadler, P.: Unraveling recursion: compiling an IR with recursion to System F. In: International Conference on Mathematics of Program Construction. pp. 414–443. Springer (2019)

[20] Krijnen, J.O.G., Chakravarty, M.M.T., Keller, G., Swierstra, W.: Translation certification for smart contracts. In: Functional and Logic Programming: 16th International Symposium, FLOPS 2022, Kyoto, Japan, May 10–12, 2022, Proceedings. p. 94. Springer, extended version available from https://arxiv.org/abs/2201.04919

[21] Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. ACM SIGPLAN Notices **49**(1), 179–191 (2014)

[22] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert—a formally verified optimizing compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress (2016)

[23] Li, J.M., Appel, A.W.: Deriving efficient program transformations from rewrite rules. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–29 (2021)

[24] Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 106–119 (1997)

[25] Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. SIGPLAN Not. **39**(4), 612–625 (Apr 2004)

[26] O'Connor, L., Chen, Z., Rizkallah, C., Jackson, V., Amani, S., Klein, G., Murray, T., Sewell, T., Keller, G.: Cogent: uniqueness types and certifying compilation. Journal of Functional Programming **31** (2021)

[27] Paraskevopoulou, Z., Eline, A., Lampropoulos, L.: Computing correctly with inductive relations. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 966–980 (2022)

[28] Park, D., Zhang, Y., Rosu, G.: End-to-end formal verification of Ethereum 2.0 deposit smart contract. In: Computer Aided Verification (CAV 2020). LNCS, vol. 12224 (2020)

[29] Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 151–166. Springer (1998)

[30] Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., Winterhalter, T.: The MetaCoq project. Journal of Automated Reasoning (2020)

[31] Spector-Zabusky, A., Breitner, J., Rizkallah, C., Weirich, S.: Total Haskell is reasonable Coq. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 14–27 (2018)